An Executable Semantics of Clock Constraint Specification Language and its Applications

Min Zhang, Frédéric Mallet

An Executable Semantics of Clock Constraint Specification Language and its Applications

Min Zhang[1] and Frédéric Mallet[2,1,3]

[1] Shanghai Key Laboratory of Trustworthy Computing
Software Engineering Institute, East China Normal University
zhangmin@sei.ecnu.edu.cn
[2] Univ. Nice Sophia Antipolis, I3S, UMR 7271 CNRS, France
Frederic.Mallet@unice.fr
[3] INRIA Sophia Antipolis Méditerranée, France

Abstract. The Clock Constraint Specification Language (CCSL) is a language to specify logical and timed constraints between logical clocks. Given a set of clock constraints specified in CCSL, formal analysis is preferred to check if there exists a schedule that satisfies all the constraints, if the constraints are valid or not, and if the constraints satisfy expected properties. In this paper, we present a formal executable semantics of CCSL in rewriting logic and demonstrate some applications of the formal semantics to its formal analysis: 1) to automatically find bounded or periodic schedules that satisfy all the given constraints; 2) to simulate the execution of schedules with customized simulation policies; and 3) to verify LTL properties of CCSL constraints by bounded model checking. Compared with other existing modeling approaches, advantages with the rewriting-based semantics of CCSL are that we do not need to assume a bounded number of steps for the formalization, and we can exhaustively explore all the solutions within a given bound for the analysis.

1 Introduction

Logical time such as defined by Lamport [9] gives a flexible abstraction to compare and order occurrences of events when appealing to more traditional physical measures is either not possible or not desirable. This is the case in a great variety of application domains, from widely distributed systems, for which maintaining a global clock can be costly, to deeply embedded software or in latency-insensitive designs [3], for which the complexity of the control mechanisms (like frequency scaling) makes it neither desirable nor efficient. In the latter case, synchronous languages [2,14] have shown that logical clocks can give a very adequate tool to represent any recurrent event uniformly, whether occurring in a periodic fashion or not.

The Clock Constraint Specification Language (CCSL) [11] is a language that handles logical clocks as first-class citizens. While synchronous languages mainly focus on signals and values and use logical clocks as a controlling mechanism, CCSL discards the values and only focuses on clock-related issues. The formal

The final pulication is available at Springer via
http://dx.doi.org/10.1007/978-3-319-29510-7_2

operational semantics of CCSL was initially defined in a research report [1] in a bid to provide a reference semantics for building simulation tools, like TimeSquare [6]. We are interested here in studying the properties of a CCSL specification and we give another formal executable semantics in rewriting logic and demonstrate the benefits of this new semantics. The first benefit is that rewriting logic gives a direct implementation of the operational semantics while TimeSquare provides a Java-based implementation, which is prone to introduce accidental complexity.

The second and most important benefit is that we can directly use rewriting logic tooling to model-check a CCSL specification. Previous works on studying CCSL properties [13], rely on several intermediate transformations to automata and other specific formats so that model-checking becomes possible when a CCSL specification is finite (or safe) [12]. It either meant, reducing to a safe subset of CCSL [8] or detecting that the specification was finite even though relying on unsafe operators. In this contribution, we rely on Maude environment [4] to provide a direct analysis support from the operational semantics and we can explore unsafe specifications by using bounded-model checking and do not restrict to the safe subset. While before, successive intermediate transformations could each introduce variations in the semantics, if not careful enough, we rely here on the strong, widely used, generic tooling provided by Maude, rather than on an ad-hoc manual implementation.

More precisely, in this paper, we introduce the notions of bounded and periodic schedules for a CCSL specification. Periodic schedules are useful to reason on specifications that rely on unsafe operators. With periodic schedules, we can use bounded model-checking to verify temporal logic properties on CCSL models. The tooling and automatic verification directly comes with the newly introduced semantics and the Maude environment.

The rest of the paper is organized as follows. Section 2 and Section 3 give a brief introduction to CCSL and Maude. In Section 4 we present the formal definition of semantics of CCSL in Maude, and in Section 5 we demonstrate four applications of the formal semantics to the analysis of CCSL. Section 6 mentions some related work and Section 7 concludes the paper.

2 CCSL

2.1 Syntax and semantics of CCSL

In CCSL, there are four primitive constraint operators which are binary relations between clocks, and five kinds of clock definitions [11]. The four constraint operators are called *precedence*, *causality*, *subclock* and *exclusion*; and the five clock definitions are called *union*, *intersection*, *infimum*, *supremum*, and *delay*.

The meaning of the nine primitive operators (see Fig. 1) is given using the notions of *schedule* and *history*. Given a set C of clocks, a schedule of C is used to decide which clocks can tick at a given step, and a history is used to calculate the number of ticks of each clock at a given step.

Definition 1 (Schedule). *Given a set C of clocks, a schedule of C is a total function $\delta : \mathbb{N}^+ \to 2^C$ such that for any n in $\mathbb{N}^+$, $\delta(n) \neq \varnothing$.*

1. $\delta \vDash c_1 \prec c_2$	$\iff \forall n \in \mathbb{N}^+.\chi(c_1,n) = \chi(c_2,n) \Rightarrow c_2 \notin \delta(n+1)$	(Precedence)
2. $\delta \vDash c_1 \preceq c_2$	$\iff \forall n \in \mathbb{N}^+.\chi(c_1,n) \geq \chi(c_2,n)$	(Causality)
3. $\delta \vDash c_1 \subseteq c_2$	$\iff \forall n \in \mathbb{N}^+.c_1 \in \delta(n) \Rightarrow c_2 \in \delta(n)$	(Subclock)
4. $\delta \vDash c_1 \# c_2$	$\iff \forall n \in \mathbb{N}^+.c_1 \notin \delta(n) \vee c_2 \notin \delta(n)$	(Exclusion)
5. $\delta \vDash c_1 \triangleq c_2 + c_3$	$\iff \forall n \in \mathbb{N}^+.(c_1 \in \delta(n) \iff c_2 \in \delta(n) \vee c_3 \in \delta(n))$	(Union)
6. $\delta \vDash c_1 \triangleq c_2 \times c_3$	$\iff \forall n \in \mathbb{N}^+.(c_1 \in \delta(n) \iff c_2 \in \delta(n) \wedge c_3 \in \delta(n))$	(Intersection)
7. $\delta \vDash c_1 \triangleq c_2 \wedge c_3$	$\iff \forall n \in \mathbb{N}^+.\chi(c_1,n) = max(\chi(c_2,n),\chi(c_3,n))$	(Infimum)
8. $\delta \vDash c_1 \triangleq c_2 \vee c_3$	$\iff \forall n \in \mathbb{N}^+.\chi(c_1,n) = min(\chi(c_2,n),\chi(c_3,n))$	(Supremum)
9. $\delta \vDash c_1 \triangleq c_2 \$ d$	$\iff \forall n \in \mathbb{N}^+.\chi(c_1,n) = max(\chi(c_2,n)-d,0)$	(Delay)

Fig. 1. Definition of 9 primitive CCSL operators

Note that a schedule must be non-trivial such that there is at least one clock ticking at any execution step. This condition excludes from schedules those steps where no clocks tick. Such steps are called *empty steps* and are trivial in that adding them to a schedule does not affect the logical relations among clocks.

Definition 2 (History). *A history of a schedule* $\delta : \mathbb{N}^+ \to 2^C$ *over a set* C *of clocks is a function* $\chi : C \times \mathbb{N} \to \mathbb{N}$ *such that for any clock* $c \in C$ *and* $n \in \mathbb{N}$*:*

$$\chi(c,n) = \begin{cases} 0 & \text{if } n = 0 \\ \chi(c,n-1) & \text{if } n \neq 0 \wedge c \notin \delta(n) \\ \chi(c,n-1)+1 & \text{if } n \neq 0 \wedge c \in \delta(n) \end{cases}$$

We use $\delta \vDash \phi$ to denote that schedule δ satisfies constraint ϕ. Fig. 1 shows the definition of the satisfiability of a constraint ϕ with regards to a schedule δ. We take the definition of precedence for example. $\delta \vDash c_1 \prec c_2$ holds if and only if for any n in $\mathbb{N}$, c_2 must not tick at step $n+1$ if the number of ticks of c_1 is equal to the one of c_2 at step n. Precedence and causality are asynchronous constraints and they forbid clocks to tick depending on what has happened on other clocks in the earlier steps. Subclock and exclusion are synchronous constraints and they force clocks to tick or not depending on whether another clock ticks or not in the same step. Union defines a clock c_1 which ticks whenever c_2 or c_3 ticks; intersection defines a clock c_1 which ticks whenever both c_2 and c_3 tick; supremum defines the slowest clock c_1 which is faster than both c_2 and c_3; infimum defines the fastest clock c_1 which is slower than both c_2 and c_3; and delay defines the clock c_1 which is delayed by c_2 with d steps. More details can be found in a recent study on CCSL [13].

Given a set Φ of clock constraints and a schedule δ, δ satisfies Φ (denoted by $\delta \vDash \Phi$) if for any ϕ in Φ there is $\delta \vDash \phi$. In particular, we use $\delta; k \vDash \phi$ to denote that δ satisfies ϕ at step $k (k \in \mathbb{N}^+)$. We use $\delta; k \vDash \Phi$ to denote that δ satisfies all the constraints in Φ at step k, i.e., $\forall \phi \in \Phi$, $\delta; k \vDash \phi$.

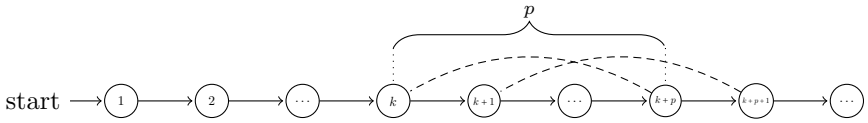

Fig. 2. Periodic schedule

2.2 Satisfiability problem of CCSL

Given a set Φ of CCSL constraints, one of the most important problems is to decide if there exist some schedules that satisfy Φ. However, it is still an open problem whether the satisfiability of a given arbitrary set of CCSL constraints is decidable or not. We consider two kinds of schedules called *bounded schedule* and *periodic schedule* from the pragmatic point of view and show the satisfiability problem of an arbitrary given set of CCSL constraints with regards to bounded schedule and periodic schedule is decidable.

Definition 3 (Bounded schedule). *Given a set Φ of clock constraints on clocks in C, and a function $\delta : \mathbb{N}_{\leq n} \to 2^C$, δ is called an n-bounded schedule if for any $i \leq n$, $\delta; i \vDash \Phi$.*

We denote the bounded satisfiability relation by $\delta \vDash_n \Phi$, which means that δ is an n-bounded schedule of Φ. It is obvious that given a bound n it is decidable to check if there exists an n-bounded schedule for a set of CCSL constraints because the number of candidate schedules is finite, i.e., $(2^{|C|} - 1)^n$, where $|C|$ denotes the number of clocks in C. If there does not exist an n-bounded schedule for a set Φ of clock constraints, there must not be a schedule that satisfies Φ, although not vice versa.

Bounded schedule is sometimes too restrictive in practice because we usually do not assign a bound to clocks in real-time embedded systems, but assume that reactive systems run forever and only terminate when shutdown. Thus, clocks should tick infinite often from the theoretical point of view. There is another class of schedules which are unbounded and force all the clocks to occur periodically. We call them *periodic schedules*.

Definition 4 (Periodic schedule). *A schedule δ is periodic if there exists k, p in $\mathbb{N}$ such that for any $k' \geq k$, $\delta(k' + p) = \delta(k')$.*

Figure 2 depicts a periodic schedule whose period is p. Each node denotes a time point, and each arrow denotes the elapse of a time unit. The dashed line indicates that, for any clock, it ticks at one point if and only if it ticks at the other point. From step k, the schedule starts to repeat every p steps infinitely. To decide whether there exists a periodic schedule for a given set of clock constraints is also an open problem. In the rest of this section, we propose an approach to constructing a periodic schedule from a bounded one when the bounded one satisfies certain conditions which are to be introduced below.

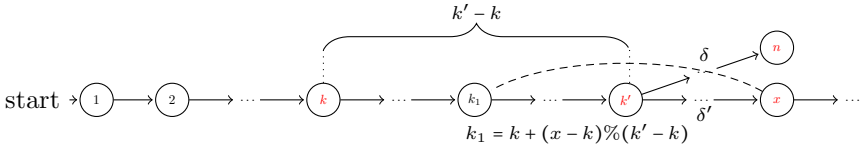

Fig. 3. Construction of periodic schedule δ' from an n-bounded schedule δ

Lemma 1. *Given a schedule $\delta : \mathbb{N}^+ \to 2^C$ and two natural numbers k, k', if there exists $m \in \mathbb{N}$ such that for any c in C $\chi(c,k)+m = \chi(c,k')$ and $\chi(c,k+1)+m = \chi(c,k'+1)$ then $\delta(k+1) = \delta(k'+1)$.*

Proof. It is equal to prove that for any $c \in C$, $c \in \delta(k+1) \iff c \in \delta(k'+1)$.

($\Rightarrow$): $c \in \delta(k+1)$ implies that $\chi(c,k+1) = \chi(c,k)+1$. Thus, $\chi(c,k+1)+m = \chi(c,k'+1) = \chi(c,k)+1+m = \chi(c,k')+1$. Thus, $c \in \delta(k'+1)$.

($\Leftarrow$): $c \in \delta(k'+1)$ implies that $\chi(c,k'+1) = \chi(c,k')+1$. Namely, $\chi(c,k+1)+m = \chi(c,k)+m+1$. Thus, $\chi(c,k+1) = \chi(c,k)+1$, and hence we have $c \in \delta(k+1)$. □

Theorem 1. *Given a schedule $\delta : \mathbb{N}^+ \to 2^C$, a clock constraint ϕ, and two natural numbers k, k', $\delta; k \vDash \phi \Rightarrow \sigma; k' \vDash \phi$ if all the following three conditions are true:*

1. *$\delta(k) = \delta(k')$;*
2. *There exists m in $\mathbb{N}$ such that $m > 0$ and for any c in C, $\chi(c,k)+m = \chi(c,k')$ and $\chi(c,k+1)+m = \chi(c,k'+1)$;*
3. *If $\phi \equiv (c_1 \triangleq c_2 \ \$ \ d)$, $\chi(c_2,k) \geq d$.*

Theorem 1 can be proved with Lemma 1. We omit the proof due to the limit of space. From Theorem 1 we can directly derive the following corollary.

Corollary 1. *Given a schedule $\delta : \mathbb{N}^+ \to 2^C$, a set Φ of clock constraints, and two natural numbers n, k', $\delta; k \vDash \Phi \Rightarrow \sigma; k' \vDash \Phi$ if the three conditions in Theorem 1 are satisfied.*

Given an n-bounded schedule δ of a set Φ of clock constraints, if there exist two natural numbers $k, k' \leq n$, which satisfy the three conditions in Theorem 1, we can define a periodic schedule δ' based on δ such that δ' satisfies Φ.

$$\delta'(x) = \begin{cases} \delta(x) & \text{if } x \leq k' \\ \delta(k + (x-k)\%(k'-k)) & \text{if } x > k' \end{cases}$$

Figure 3 shows the construction of δ' based on δ. From k', the schedule δ' repeats infinitely the steps from k to $k'-1$. By Corollary 1, it is obvious that for any k'' such that $k'' > k'$, we have $\delta'; k'' \vDash \Phi$ because we can find a natural number $k_1 = k + (k''-k)\%(k'-k)$ such that $\delta; k_1 \vDash \Phi$, $\delta(k_1) = \delta'(k_1)$ and k'', k_1, δ' satisfy the three conditions in Theorem 1. Thus, we have $\delta' \vDash \Phi$.

3 Maude in a Nutshell

Maude is rewriting-based algebraic language and also an efficient rewriting engine. We assume the readers are familiar with Maude, and only give a brief introduction to Maude meta-level functionality and Maude LTL model checking, which is used in this paper. More details about Maude can be found in the Maude book [4].

The underlying logic of Maude is rewriting logic, which is reflective in the sense that it can be faithfully interpreted in itself [4]. The reflectivity allows us to reason with a specified rewrite theory in customized strategies by Maude. Intuitively, we define a rewrite theory $\mathcal{R}$ and then define a metatheory $\mathcal{U}$ where $\mathcal{R}$ is treated as data. A rewrite theory $\mathcal{R}$ is a tripe $\langle \Sigma, E, R \rangle$, where Σ is called the signature specifying the type structure, E is a set of equations and R is a set of rewrite rules. Maude provides efficient function by command `search` to find if there exist some pathes from a given term t to a target term t' by repeatedly applying the rewrite rules in R. It also provides a corresponding meta-level searching function `metaSearch` which takes $\mathcal{R}$, t and t' as arguments and returns the searching result. An LTL model checker has been implemented based on Maude to verify LTL properties of a rewrite theory when the set of states that are reachable from an initial state in the rewrite theory is finite [7].

4 Formal Semantics of CCSL in Maude

We formalize a clock as a triple (c, ℓ, n), consisting of the clock identifier c, a list ℓ of records, with each value being *tick* or *idle* (abbreviated by t or i respectively), representing that the clock ticks or not at the corresponding step, and a natural number n to indicate the numbers of ticks in ℓ. ℓ represents a bounded schedule of c whose bound is equal to the length of ℓ. Initially, ℓ is empty and n is 0. Let $\mathcal{C}$ be the set of such clock triples of a set C of clocks. We call $\mathcal{C}$ a *configuration*. We suppose that the length of the lists in each clock triple in $\mathcal{C}$ are equal, e.g. n. $\mathcal{C}$ essentially represents an n-bounded schedule for all the clocks in C.

We declare a predicate `satisfy` which takes three arguments: a configuration $\mathcal{C}$, a non-zero natural number k, and a set Φ of constraints, and returns true if $\mathcal{C}$ satisfies Φ at step k, and otherwise false. We consider each possible constraint form in Φ when defining `satisfy`. For instance, the following two equations are defined to specify a configuration $\mathcal{C}$ satisfies precedence and infimum at step k:

```
ceq satisfy(C, k, c1 < c2) = (num(l1,k) >= num(l2, k)) and
    (if num(l1,k-1) == num(l2, k-1) then t-val(l2,k) =/= t else true fi)
 if (c1, l1, n1) := getConf(C, c1) /\ (c2, l2, n2) := getConf(C, c2).
ceq satisfy(C, k, c1 ≜ c2 ∧ c3) = (if n2 > n3 then n1 == n2 else n1 == n3 fi)
 if (c1, l1, n1) := getConf(C, c1) /\ (c2, l2, n2) := getConf(C, c2)
   /\ (c3, l3, n3) := getConf(C, c2) .
```

The first equation says that `satisfy` returns true with $\mathcal{C}$, k and $c_1 \prec c_2$ when the number of ticks of c_1 up to step k is greater than or equal to the one of c_2 and further if the number of ticks of c_1 up to step $k-1$ is the same as the one of c_2 then

c_2 must not tick at step k (as represented by `t-val(`ℓ_2`,`k`) =/=` t, where `t-val` is a function returning the k^{th} value in the list ℓ_2). The equation has a condition which is a conjunction of two matching equations [4]. The two conjuncts are used to retrieve the tick list and the number of ticks of c_1 (and c_2) by function `getConf` and assign them to ℓ_1 and n_1 (and ℓ_2 and n_2). The second equation defines the semantics of infimum relation, namely, at any step k the number of ticks of c_1 must be the minimum of those of c_2 and c_3. The correspondence between the formalization of the constraints and their formal semantics defined in Figure 1 should be clear. Other constraints can be formalized in Maude likewise, and we omit them from the paper.

Next we formalize one-step ticking from k to $k+1$ of all clocks by a set of rewrite rules. The basic idea is as follows. From step k to $k+1$ each clock decides to tick or not (be idle). After all the clocks make a decision, we check if the bounded schedule satisfies all the constraints at step $k+1$. The first rewrite rule at Line 1 specifies the behavior that clock c ticks at step $k+1$. The list ℓ is changed into $\ell\ t$. The rule is conditional because we need the condition that c is not the last clock which makes a decision. If c is the last one, we need to check if all the constraints in Φ are satisfied at step $k+1$. The step k can be represented by the length of the list ℓ of an arbitrary clock triple in $\mathcal{C}$, i.e., k = `size(`ℓ`)`, where `size(`ℓ`)` returns the length of ℓ. Thus, $k+1$ is equal to `size(`ℓ`)` + 1, and hence we use the latter one in the condition of the fourth equation on Line 6.

Similarly, if c decides to remain idle next step and c is not the last clock, its corresponding tick list is changed from ℓ to $\ell\ i$, which is specified by the rule on Line 2. If c is the last clock in this case, we also need to guarantee that from step k to $k+1$ there must be at least one clock ticking (represented by the formula `not allIdle(`$\mathcal{C}'$`)`) and all the clocks satisfy the constraints in Φ at step $k+1$.

```
1 crl ((c,ℓ,n) C ; C' ; Φ) => (C ; C' (c,ℓ t,n+1) ; Φ) if C =/= nil .
2 crl ((c,ℓ,n) C ; C' ; Φ) => (C ; C' (c,ℓ i,n) ; Φ)    if C =/= nil .
3 crl ((c,ℓ,n) ; C' ; Φ)   => (nil ; C' (c,ℓ t,n+1) ; Φ )
4     if satisfy(C' (c,ℓ t,n+1), size(ℓ) + 1, Φ) .
5 crl ((c,ℓ,n) ; C' ; Φ)   => (nil ; C' (c,ℓ i,n) ; Φ)
6     if not allIdle(C') /\ satisfy(C' (c,ℓ i,n), size(ℓ) + 1, Φ) .
```

We assume that $\mathcal{C}$ is a k-bounded schedule of a set Φ of CCSL constraints. If there is a rewriting sequence from `(`$\mathcal{C}$`; nil; `Φ`)` to a new one `(nil; `$\mathcal{C}'$`; `Φ`)` with the above four rules, $\mathcal{C}'$ must be a $k+1$-bounded schedule of Φ because $\mathcal{C}'$ satisfies Φ up to $k+1$ steps. We can define the following rule to specify the one-step ticking of all the clocks from step k to $k+1$.

```
1 crl < C ; k ; Φ > => < C' ; k+1 ; Φ > if (C ; nil ; Φ) => (nil ; C' ; Φ) .
```

The condition of the rule is a *rewrite condition* [4], which is true if and only if there exists a rewriting sequence from the term at the left-hand side of `=>` to the one at the right-hand side when the condition is true. In the above rule, $\mathcal{C}'$ represents an arbitrary immediate successor of $\mathcal{C}$ such that $\mathcal{C}'$ satisfies Φ up to $k+1$ steps.

5 Applications of the Formal Semantics

In this section, we show four applications of the executable formal semantics of CCSL in Maude.

5.1 Bounded scheduling

Given a bound n and a set of clock constraints Φ, we can use Maude's `search` function to find automatically if there exists an n-bounded schedule of Φ. If Maude cannot find a schedule within a given bound n, it means that there must not exist such an n-bounded schedule, and further we can conclude that there must not exist a schedule that satisfies Φ. However, if a schedule is found up to bound n, we only can conclude that the returned schedule is n-bounded, but cannot guarantee the existence of a schedule for Φ.

We show an example of finding bounded schedules for a given set of clock constraints using Maude's `search` command.

Example 1. Given a set of constraints $\Phi_1 = \{c_1 \prec c_2, c_3 \triangleq c_1 \$ 1\ , c_2 \prec c_3$, we can use Maude's `search` command to find a 100-bounded schedule.

```
search [2,100] init(Φ1) =>* < C ; 100 ; (Φ1) > .
states: 101  rewrites: 629424
C -->
  (c1,t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t
      i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i
      t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i,50)
  (c2,i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i
      t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t
      i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t,50)
  (c3,i i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t
      i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i
      t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i,49)
```

Maude's `search` command takes two optional arguments in the square brackets. The first one is used to specify the number of expected solutions, and the second one is used to specify the maximal depth of searching. Function `init` takes a set Φ of constraints and generates an initial configuration `< C0 ; 0 ; Φ >`, where $\mathcal{C}_0$ is a set of clock triples, each of which is of the form $(c, \texttt{nil}, 0)$. The operator `=>*` indicates there are zero or more rewritings from the given initial term to the expected term that can be matched by the target term.

In this example, the target term represents those configurations where the current step is 200. $\mathcal{C}$ is a configuration which is assigned by Maude. The result is obtained by repeatedly applying the rewrite rule. Maude only returns one result with the command. It means that there is only one possible 100-bounded schedule for the constraints. The schedule shows that c_1 and c_3 only tick at all odd steps except that c_3 does not tick at the first step, because of the constraint $c_3 \triangleq c_1 \$ 1\ .\ c_2$ only ticks at all even steps. The returned bounded schedule coincides with the result in an earlier work of the second author [13].

5.2 Customized simulation

Given a set Φ of clock constraints, it is also desirable to have a customized schedule which satisfies not only Φ but also some customized requirements, e.g., at each step if a clock can tick it must tick, or if a clock does not have to tick, it must not tick. We only consider three basic scheduling policies, called *randomness*, *maximum* and *minimum* respectively.

- *Randomness*: If a clock can tick and not tick at next step, we randomly choose one.
- *Maximum*: If a clock can tick at next step, it must tick.
- *Minimum*: If a clock has not to tick at next step, it must not tick.

Based on the four rewrite rules defined in Section 4, we can achieve customized scheduling for a given set of clock constraints using Maude's meta-level facility. We first find all the possible immediate successors of a set $\mathcal{C}$ of clock triples using Maude's `metaSearch` function, and then choose the successor that satisfies the customized policy given by users. The following rewrite rule is defined for customized scheduling.

```
--- the rewrite rule is defined for customized scheduling
crl < C ; k ; Φ ; ρ > => < C' ; k+1 ; Φ ; ρ > if C' := conf(sucs(C, Φ), ρ) .
--- the equation needs the meta-level function metaSearch to compute all successors
ceq sucsAux(C, Φ, j) = downTerm(T, nil), sucsAux(C, Φ, j + 1)
 if RT := metaSearch(upModule('ONE-STEP-TICKING, false),
           ''(_;_;_')[upTerm(C), 'nil.Conf, upTerm(Φ)],
           ''(_;_;_')['nil.Conf, C', upTerm(Φ)],nil,'*,unbounded,i) /\
    (C' <- T) := getSubstitution(RT) .
```

In the rule, ρ is a variable, denoting the customized policy given by users, e.g. `rand` for randomness, `max` for maximum or `min` for minimum. The function `sucs` used in the condition returns the set of all the successors of $\mathcal{C}$ that satisfy Φ, and `conf` returns one among them according to the customized policy ρ. The equation above is used to define a recursive function `sucsAux`, which is the main auxiliary function to define `sucs`. Function `sucsAux` takes three arguments, $\mathcal{C}$, Φ and a natural number j, which indicates that we want `metaSearch` to find the $j^{\text{th}}(j \geq 0)$ successor of $\mathcal{C}$. The `metaSearch` function takes a meta-module of the module `ONE-STEP-TICKING` where the four rewrite rules in Section 4 are defined, a term from which searching begins, a target term that the result term can match, and other three arguments, and returns a searching result. The searching result contains a meta-level term which substitute for $\mathcal{C}'$. We change it to the object level by the built-in function `downTerm`. The object-level term represents the i^{th} successor of $\mathcal{C}$. We omit the detailed explanation about the usage of `metaSearch`. Interested readers can refer to the work [4] for the details.

Example 2. Let Φ_2 be the set of the following constraints:

$$in_1 \preccurlyeq step_1 \qquad step_1 \prec step_3 \qquad in_2 \preccurlyeq step_2$$
$$step_2 \prec step_3 \qquad step_3 \preccurlyeq out$$

We show the simulations of the bounded schedules that satisfy Φ_2 with maximum and minimum policy. We use Maude's `rew` command to rewrite the initial configuration $< \mathcal{C}_0 \ ; \ 0 \ ; \ \Phi \ ; \ \rho >$ by applying the rewrite rule defined in this section 10 times with `max` and `min` respectively. The initial configuration is generated by function `init1`, which takes a set Φ of CCSL constraints and a simulation policy ρ as its arguments. The commands and returned results are shown as follow.

```
rew [10] init1(Φ2, max) .
result CCC: ('in1,  t t t t t t t t t t,10)('in2,  t t t t t t t t t t,10)
            ('out,  i t t t t t t t t t,9) ('step1,t t t t t t t t t t,10)
            ('step2,t t t t t t t t t t,10)('step3,i t t t t t t t t t,9)...
rew [10] init1(Φ2, min) .
result CCC: ('in1,  i i i i i i i i i i,0) ('in2,  t i t i t i t i t i,5)
            ('out,  i i i i i i i i i i,0) ('step1,i i i i i i i i i i,0)
            ('step2,i t i t i t i t i t,5) ('step3,i i i i i i i i i i,0)...
```

For the first schedule, the number of ticking clocks is always maximal, while for the second one the number of ticking clocks is always minimal.

5.3 Periodic scheduling

We also can find automatically periodic schedules of a given set of CCSL constraints by Maude's `search` command with the rewriting-based semantics of CCSL in Maude. The basic idea is to compute all possible immediate successors of the current k-bounded schedule at every step $k(k \geq 1)$ and check if there exists a successor that satisfies all the three conditions in Theorem 1. If such a successor exists, a periodic schedule is found, and the step $k+1$ is the first step of the second iteration. We also can compute the period of the schedule. The following rewrite rule is defined for periodic scheduling.

```
--- the rewrite rule is defined to represent periodic schedules
crl < C ; k ; Φ ; 0 > =>
    if C'' == nil then < C' ; k+1 ; Φ ; 0 > else < C'' ; k+1 ;  Φ ; p > fi
 if (CS1,C',CS2) := sucs(CF,CTS) /\ <C''; p> := checkOcc((CS1, C',CS2),k+1) .
```

The terms at left-hand side of the rule is a 4-tuple, where the last argument, i.e., 0 is indicates that the k-bounded schedule does not satisfy the three conditions in Theorem 1. Function `checkOcc` is used to check if there exists a $k+1$-bounded schedule that satisfies all the constraints in Φ and also the three conditions in Theorem 1. If that is the case, `checkOcc` returns the schedule $\mathcal{C}''$ and the period $p(p > 0)$, and otherwise nil and 0. Once a periodic schedule is found, the rewrite rule cannot be applied and Maude returns the result. Note that the rule may cause non-termination if no periodic schedule is found and no bound to the times of rewriting is set.

As an example, we use Maude's `search` command to find periodic schedules of the precedence constraint $c_1 \prec c_2$. The command is as follows:

```
search [4] init2(c1 < c2) =>* < C; k ; c1 < c2 ; p > such that p =/= 0 .
```

Function `init2` takes a set Φ of CCSL constraints and returns an initial configuration $< \mathcal{C}_0 \ ; \ 0 \ ; \ \Phi \ ; \ 0 >$, where the last natural number is used to record the

Table 1. Four periodic schedules that satisfy $c_1 \prec c_2$

schedule	clock/step	1	2	3	4	5	6	...	period p
1	c_1	t	t	t	t	t	t	...	1
	c_2	i	t	t	t	t	t	...	
2	c_1	t	i	t	i	t	i	...	2
	c_2	i	t	i	t	i	t	...	
3	c_1	t	t	t	t	t	t	...	1
	c_2	i	i	t	t	t	t	...	
4	c_1	t	t	t	i	t	i	...	2
	c_2	i	i	i	t	i	t	...	

period of the current bounded schedule. We provide an upper bound e.g. 4 to the expected periodic schedules. In the command, $\mathcal{C}$ is a set of two clock triples of c_1 and c_2 returned by Maude when a periodic schedule is found. k indicates the step where the first period of the schedule ends, and p indicates the period of the schedule. The condition p `=/=` 0 means that $\mathcal{C}$ represents a periodic schedule. Table 1 shows four periodic schedules found by Maude for $c_1 \prec c_2$ when the bound is set 4. The red steps for each schedule are the beginning of the first and second iteration of the period. We also can give p a concrete value and use Maude to search those periodic schedules with a fixed period.

5.4 Bounded model checking

Given a set of clock constraints, it is desired to know if the constraints satisfy some properties, e.g. if all the clocks can tick infinitely often, or a clock must tick immediately after another clock ticks. Based on the formal semantics of CCSL in Maude, we can model check LTL properties of a given set of CCSL constraints by Maude LTL model checker. Maude model checker requires the reachable state space being verified must be finite, while the reachable state space specified by the rewrite theory of a set of clock constraints may be infinite if there exists some non-periodic schedules. For periodic schedules, we force the schedule to repeat from step n to n' where n and n' are the beginning and ending steps of the first period. As depicting by Fig. 4, by setting a bound we can compute all periodic schedules up to the bound. The periodic schedules compose a finite state space which can be used for model checking. Figure 4 (left) shows an example of an unbounded state space. Each path represents a schedule. The path with a loop represents a periodic schedule. There are three periodic schedules in the figure when the bound is set 3. The three periodic schedules constitute a finite state space which can be model checked, as shown in Figure 4 (right).

Next, we show some basic properties that clock constraints are expected to satisfy and their representations in LTL formula. Let *tick* be a parameterized predicate on states, which takes a clock c as argument and returns true if c ticks in a state and otherwise false.

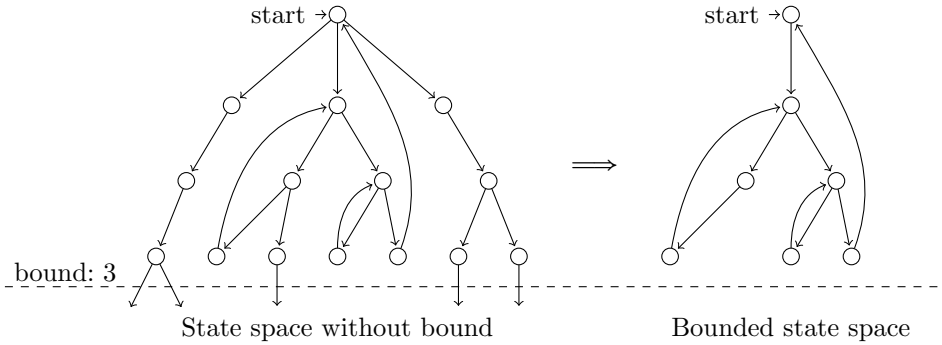

Fig. 4. Bounded state space of periodic schedulers

- *Repeated ticking*: all clocks must tick infinitely often, which can be formalized as: $\bigwedge_{c \in C} \Box \Diamond tick(c)$.
- *Simultaneous ticking*: two clocks c_1 and c_2 must tick simultaneously, which can be formalized as: $\Box(tick(c_1) \iff tick(c_2))$.
- *Leading-to ticking*: if a clock c_1 ticks, it must cause another clock c_2 to tick eventually, which can be formalized as: $\Box(tick(c_1) \rightarrow \Diamond tick(c_2))$.
- *Alternating ticking*: two clocks c_1 and c_2 must always tick immediately after each other, which can be formalized as: $\Box(tick(c_1) \rightarrow \bigcirc tick(c_2) \wedge tick(c_2) \rightarrow \bigcirc tick(c_1))$.

As an example, we model check if the constraints Φ_1 in Example 1 satisfy the alternating ticking property.

```
--- definition of the state predicate tick in Maude
ceq <(C; k; Φ; p> |= tick(c) = (tval(ℓ,k) == t) if (c,ℓ,n) := getConf(C, c) .
--- the following command is used for model checking in Maude
red modelCheck(init2((c1 ≺ c2)(c3 ≜ c1 $ 1 )(c2 ≺ c3)),
    [](tick(c1) -> O tick(c2) /\ tick(c2) -> O tick(c1))) .
Result: true
```

The first equation is used to define the state predicate `tick`, and `modelCheck` is a built-in function to do model checking in Maude. It takes an initial state (configuration) and an LTL formula. Maude returns true with the above command, which means that the constraints Φ_1 indeed satisfies the alternating ticking property. This result coincides with the one obtained by encoding CCSL into finite-state transition system [13].

By bounded model checking in Maude we also can find invalid schedules of a given set of clock constraints. A schedule is called invalid if it prevent some clock from ticking after some step, namely, it does not satisfy the repeated ticking property. A set Φ of CCSL constraints are called invalid if there exist invalid schedules that satisfy Φ. Once Maude finds such a periodic schedule that violates the repeated ticking property, we can conclude that the constraints

Table 2. Eight deadlock schedules found by Maude for CCSL constraints Φ_2'

No.	in_1	in_2	$step_1$	$step_2$	$step_3$	out	tmp_1	tmp_2
1	t	i	t	i	i	i	t	i
2	i	t	i	t	i	i	t	i
3	$t\ i$	$i\ i$	$i\ t$	$i\ i$	$i\ i$	$i\ i$	$t\ i$	$i\ i$
4	$i\ i$	$t\ i$	$i\ i$	$i\ t$	$i\ i$	$i\ i$	$t\ i$	$i\ i$
5	$t\ i\ t$	$t\ i\ i$	$t\ i\ t$	$t\ i\ i$	$i\ t\ i$	$i\ t\ i$	$t\ i\ t$	$i\ i\ t$
6	$t\ i\ i$	$t\ i\ t$	$t\ i\ i$	$t\ i\ t$	$i\ t\ i$	$i\ t\ i$	$t\ i\ t$	$i\ i\ t$
7	$t\ i\ t$	$t\ i\ i$	$t\ i\ t$	$i\ t\ i$	$i\ t\ i$	$i\ t\ i$	$t\ i\ t$	$i\ i\ t$
8	$t\ i\ i$	$t\ i\ t$	$t\ i\ i$	$i\ t\ t$	$i\ t\ i$	$i\ t\ i$	$t\ i\ t$	$i\ i\ t$

are not valid. However, it cannot grantee the constraints are valid if no invalid schedules are found because not all schedules are model checked.

A special invalid case of CCSL constraints is that some schedules may prevent all clocks from ticking after some step. We call them *deadlock schedules*. We can use Maude to find if there exist deadlock schedules within a given bound. Let us consider a case of Example 2. Assume that we introduce the following four new constraints to Φ_2 and denote the new set as Φ_2':

$$tmp_1 \triangleq in_1 + in_2 \qquad tmp_1 \prec out \qquad tmp_2 \triangleq tmp_1 \$ 1 \qquad out \prec tmp_2$$

The four constraints mean that clocks tmp_1 and out must alternatively tick. We can find a number of schedules satisfying all the constraints in Φ_2'. However, some of them may cause deadlock. We find 8 deadlock schedules by searching within 3 steps in Maude with the command:

```
search [10,3] init(Φ'2)=>! CF .
```

In the command $\mathcal{CF}$ is a variable to which a 4-tuple is going to be assigned by Maude, and `=>!` means that the value assigned to $\mathcal{CF}$ must be rewritten by any rewrite rules. Namely, the value assigned to $\mathcal{CF}$ is a deadlock schedule. Table 1 shows the eight deadlock schedules. We take the first one as an example. According to the first schedule, only three clocks, i.e. in_1, $step_1$ and tmp_1 tick at the first step. In next step, no clocks can tick because of the newly introduced four constraints. For instance, in_2 cannot tick in next step. If in_2 ticked, so did tmp_1 (by constraint $tmp_1 \triangleq in_1 + in_2$) and tmp_2 (by constraint $tmp_2 \triangleq tmp_1 \$ 1$), which violates the constraint $out \prec tmp_2$. Because in_2 cannot tick, $step_2$ can neither by constraint $in_2 \prec step_2$. Other clocks also cannot tick because of the corresponding constraints, leading to a deadlock.

6 Related Works and Discussion

CCSL mainly deals with logical clocks, i.e., unbounded increasing sequences of integers. The semantics of clock constraints may depend on boolean parameters,

in which case, we remain in a finite world and can rely on traditional verification and analysis results and tools. The constraints may also depend on unbounded integer values, for instance, the number of times a given clock has ticked. In this latter case, the constraint is called unsafe [12]. A specification is safe if it does not use any unsafe constraint.

The reference semantics of CCSL was given in a research report [1] mainly to be able to define a simulation tool called TimeSquare [6]. TimeSquare encodes the operational semantics of CCSL in Java and captures boolean constraints symbolically using Binary Decision Diagrams (BDD). TimeSquare works step by step and at each step, finding a solution reduces to a satisfiability problem. After deciding if and how many valid solutions can be found at a step, TimeSquare clock engine picks one solution according to its simulation policy, updates the state space and moves forward. TimeSquare does not consider the unbounded specification as a whole and only produce one finite possible trace that satisfies all the constraints up to a given number of steps. In this work, we use bounded model-checking, we can then explore all the solutions reached in a given number of steps, instead of only one.

Other works have tried to make an exhaustive exploration of the entire state space (not up to a pre-defined number of steps). A comprehensive list of such works has been summarized in a recent survey [13]. However, one aspect is to be able to decide whether the state space can be represented with a finite abstraction even though the specification is unsafe. Another way is to force a finite space space by restricting to safe constraints [16,8,15]. In this work, we do not make any assumptions on whether the specification is safe or not.

The most important achievement in this paper is that, thanks to Maude environment, all the analyses performed result directly from the operational semantics without intermediate transformations, so without the need to prove that the semantics is preserved. Yu et al. proposed to encode CCSL in Signal before transforming it to the internal format of Sigali [16]. We hope that the encoding in Maude will allow to conduct automated verification for all the transformational approaches that use CCSL as a step. Maude also gives a framework to define the simulation policies formally. Some undocumented simulation policies are available in TimeSquare [6]. In Section 4, we give a simple formal interpretation for three of these simulation policies.

Finally, abstract interpretation [5] or infinite model-checking [10] would allow reasoning on the global CCSL specification without restrictions. However, the encoding is likely to introduce semantic variations and we do not know at the moment how to encode CCSL constraints in a compositional way.

7 Conclusion and Future Work

We have proposed a new semantic model for CCSL constraints. We have also introduced the notion of bounded and periodic schedules. The satisfiability problem for CCSL specifications, which is still an open problem in the general case, is proved to be decidable with regards to bounded and periodic schedules even

when using unsafe constraints. This is the first main result. The second result is to use the Maude encoding to perform bounded scheduling, customized simulation with different policies, periodic scheduling, and bounded model-checking.

The notion of periodic schedule seems promising but a bit constraining. In the future, we shall try to provide a more general definition where the behavior might slightly vary between successive periods while still maintaining decidability.

References

1. André, C.: Syntax and semantics of the Clock Constraint Specification Language (CCSL). Research Report 6925, INRIA (2009)
2. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Le Guernic, P., de Simone, R.: The synchronous languages 12 years later. Proceedings of the IEEE 91(1), 64–83 (2003)
3. Carloni, L.P., McMillan, K.L., Sangiovanni-Vincentelli, A.L.: Theory of latency-insensitive design. IEEE Trans. on CAD of Integrated Circuits and Systems 20(9), 1059–1076 (2001)
4. Clavel, M., et al.: All about Maude. LNCS 4350, Springer (2007)
5. Cousot, P.: Abstract interpretation. ACM Comput. Surv. 28(2), 324–328 (1996)
6. Deantoni, J., Mallet, F.: Timesquare: Treat your models with logical time. In: Furia, C.A., Nanz, S. (eds.) TOOLS (50). Lecture Notes in Computer Science, vol. 7304, pp. 34–41. Springer (2012)
7. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL model checker. In: 4th WRLA, ENTCS 71. pp. 162–187. Elsevier (2002)
8. Gascon, R., Mallet, F., DeAntoni, J.: Logical time and temporal logics: Comparing UML MARTE/CCSL and PSL. In: Combi, C., Leucker, M., Wolter, F. (eds.) TIME. pp. 141–148. IEEE (2011)
9. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978)
10. Leroux, J., Sutre, G.: Flat counter automata almost everywhere! In: 3rd International Symposium on Automated Technology for Verification and Analysis. Lecture Notes in Computer Science, vol. 3707, pp. 489–503. Springer (2005)
11. Mallet, F., André, C., de Simone, R.: CCSL: specifying clock constraints with UML/Marte. Innovations in Systems and Software Engineering 4(3), 309–314 (2008)
12. Mallet, F., Millo, J.V., de Simone, R.: Safe CCSL specifications and marked graphs. In: 11th ACM/IEEE Int. Conf. on Formal Methods and Models for Codesign. pp. 157–166. IEEE (2013)
13. Mallet, F., de Simone, R.: Correctness issues on MARTE/CCSL constraints. Sci. Comput. Program. 106, 78–92 (2015)
14. Potop-Butucaru, D., de Simone, R., Talpin, J.: The Synchronous Hypothesis and Polychronous Languages, chap. 6. CRC Press (2009)
15. Yin, L., Mallet, F., Liu, J.: Verification of MARTE/CCSL time requirements in Promela/SPIN. In: Perseil, I., Breitman, K., Sterritt, R. (eds.) ICECCS. pp. 65–74. IEEE Computer Society (2011)
16. Yu, H., Talpin, J., Besnard, L., Gautier, T., Marchand, H., Guernic, P.L.: Polychronous controller synthesis from MARTE/CCSL timing specifications. In: 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE. pp. 21–30. IEEE (2011)